Overview

- More uninformed search: depth-limited, iterative deepening, bidirectional search
- Avoiding repeated state
- Constraint satisfaction search
- Informed search: domain knowldege to evaluation function

Seminar Credits

2% Extra credit (upto 4 talks, i.e. a total of 8%) for attending the selected (I will announce which one's eligible) seminars and submitting:

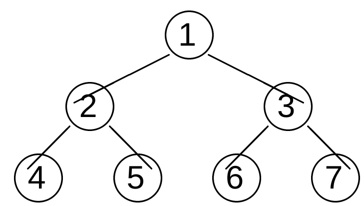
1. one paragraph summary of the talk (don't copy the abstract - focus on the part that was the most interesting to you)
2. pros and cons of the approach (one paragraph)
3. your idea on how to solve the problem (one paragraph)
4. send it to **choe@tamu.edu** via email.

* If you find any other talk at TAMU that is related to AI of intelligence, ask me if you can earn the extra credit by attending the talk.

Distinguished Lecturer

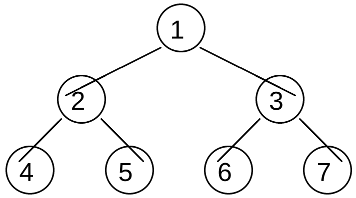
- Dr. Kimon Valavanis
- Evolutionary Algorithm Based Off-line/On-line Path Planner for UAV Navigation
- 4:10pm, Today (1/30/02)
- HRBB Room 124

Depth Limited Search (DLS): Limited Depth DFS



- node visit order for each depth limit l:
  1 ($l = 1$); 1 2 3 ($l = 2$); 1 2 3 4 5 6 7 ($l = 3$);
- queuing function: enqueue at front (i.e. stack push)
- **push the depth of the node as well**:
  ($<$depth$>$ $<$node$>$)

DLS: Expand Order



Evolution of the queue (**bold**=expanded and then added): (<depth>,<node>)); Depth limit = 3

1. [(d1,1)] : initial state
2. **[(d2,2)][(d2,3)]** : pop 1 and push 2 and 3
3. **[(d3,4)][(d3,5)]**[(d2,3)] : pop 2 and push 4 and 5
4. [(d3,5)][(d2,3)]: pop 4, cannot expand it further
5. [(d2,3)]: pop 5, cannot expand it further
6. **[(d3,6)][(d3,7)]**: pop 3, and push 6, 7

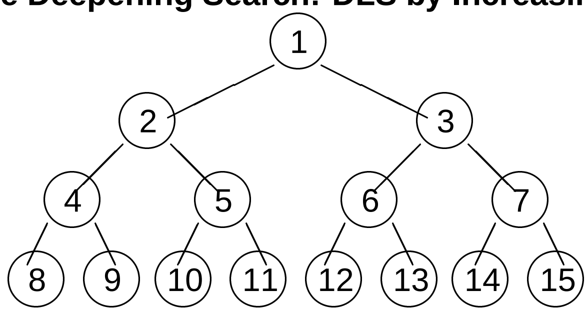
...

DLS: Evaluation

branching factor b, depth limit l, depth of solution d:

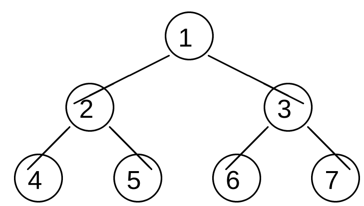
- complete: if $l \geq d$
- time: b^l nodes expanded (worst case)
- space: bl (same as DFS, where $l = m$ (m: max depth of tree in DFS)
- good if solution is within the limited depth.
- non-optimal (same problem as in DFS).

Iterative Deepening Search: DLS by Increasing Limit



- node visit order:
  1 ; 1 2 3; 1 2 3 4 5 6 7; 1 2 3 4 5 8 9 10 11 6 7 12 13 14 15; ...
- revisits already explored nodes at successive depth limit
- queuing function: enqueue at front (i.e. stack push)
- **push the depth of the node as well**: **(<depth> <node>)**

IDS: Expand Order



Basically the same as DLS: Evolution of the queue (**bold**=expanded and then added): (<depth>,<node>)); e.g. Depth limit = 3

1. [(d1,1)] : initial state
2. **[(d2,2)][(d2,3)]** : pop 1 and push 2 and 3
3. **[(d3,4)][(d3,5)]**[(d2,3)] : pop 2 and push 4 and 5
4. [(d3,5)][(d2,3)]: pop 4, cannot expand it further
5. [(d2,3)]: pop 5, cannot expand it further
6. **[(d3,6)][(d3,7)]**: pop 3, and push 6, 7

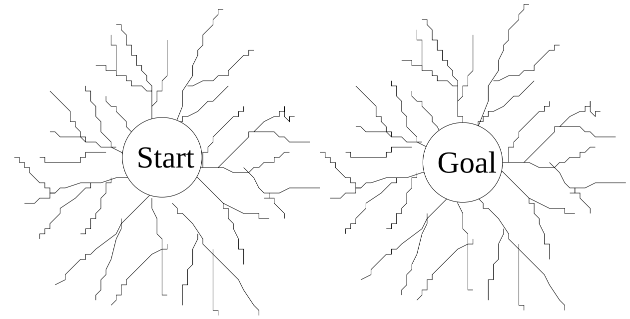
...

IDS: Evaluation

branching factor b, depth of solution d:

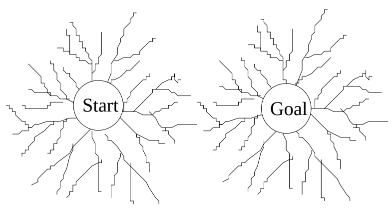
- complete: cf. DLS, which is conditionally complete
- time: b^d nodes expanded (worst case)
- space: bd (cf. DFS and DLS)
- **optimal!**: unlike DFS or DLS
- good when search space is huge and the depth of the solution is not known (*)

Bidirectional Search (BDS)



- Search from both initial state and goal to reduce search depth.
- $O(b^{d/2})$ of BDS vs. $O(b^d)$ of BFS.

BDS: Considerations



1. how to back trace from the goal?
2. successors and precedecessors: are operations reversible?
3. are goals explicit?: need to know the goal to begin with
4. check overlap in two branches
5. BFS? DFS? which strategy to use? Same or different?

BDS Example: 8-Puzzle

5	4	
6	1	8
7	3	2

→

5	4	8
6	1	
7	3	2

→ ... ←

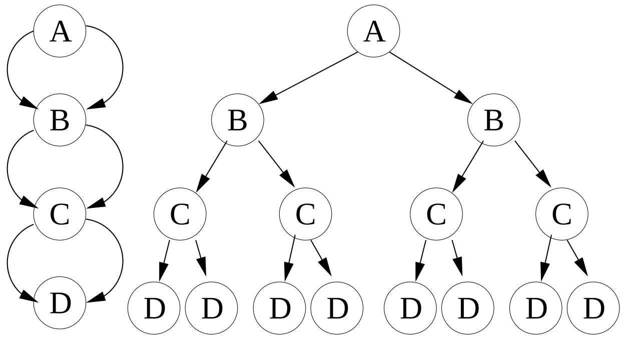
1	2	3
	8	4
7	6	5

←

1	2	3
8		4
7	6	5

- Is it a good strategy?
- What about Chess? Would it be a good strategy?
- What kind of domains may be suitable for BDS?

Avoiding Repeated States



Repeated states can be devastating in search problems.

- Common cases: problems with reversible operators → search space becomes infinite
- One approach: find a spanning tree of the graph

Avoiding Repeated States: Strategies

5	4	
6	1	8
7	3	2

→

5	4	8
6	1	
7	3	2

→

5	4	
6	1	8
7	3	2

→

5	4	8
6	1	
7	3	2

...

- Do not return to the node's parent
- Avoid cycles in the path (this is a huge theoretical problem in its own right)
- Do not generate states that you generated before: use a hash table to make checks efficient

How to avoid storing every state? Would using a short signature (or a checksum) of the full state description help?

Constraint Satisfaction Search

Constraint Satisfaction Problem (CSP):

- **state:** values of a set of **variables**
- **goal**: test if a set of constraints are met
- **operators**: set values of variables
- general search can be used, but specialized solvers for CSP work better

Constraints

- Unary, binary, and higher order constraints: how many variables should simultaneously meet the constraint
- Absolute constraints vs. preference constraints
- Variables are defined in a certain**domain**, which determines the possible set of values, either discrete or continuous.

This is part of a much more complex problem called **constrained optimization problems** in operations research consisting of cost function (either minimize or maximize) and several constraints. Problems can be linear, nonlinear, convex, nonconvex, etc. Straight-forward solutions exist for a limited subclass of these (for example, for linear programming problems can be solved by the simplex method).

CSP: continued

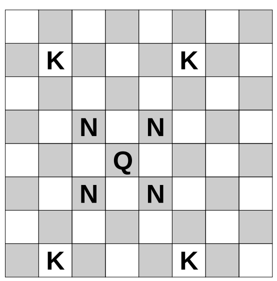
- CSPs include NP-complete problems such as 3-SAT, thus finding the solutions can require exponential time.
- However, constraints can help narrow down the possible options, therefore reducing the branching factor. This is because in CSP, the goal can be decomposed into several constraints, rather than being a whole solution.
- Strategies: backtracking (back up when constraint is violated), forward checking (do not expand further if look-ahead returns a constraint violation). Forward checking is often faster and simple to implement.

Informed Search (Chapter 4)

From domain knowledge, obtain an **evaluation function**.

- best-first search: order nodes according to the evaluation function value
- greedy search: minimize estimated cost for reaching the goal – fast, but incomplete and non-optimal.
- A^*: minimize $f(n) = g(n) + h(n)$, where $g(n)$ is the current path cost from start to n, and $h(n)$ is the estimated cost from n to goal.

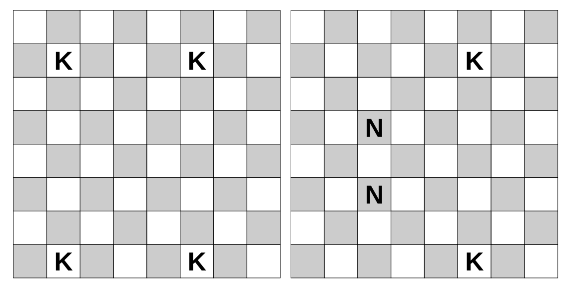
When Greedy Search Fails



- Remove minimum number of pieces so that no piece is attacked
- Greedy strategies: most attacked, most attacking, or sum of both, etc.

Is there any other greedy strategy?

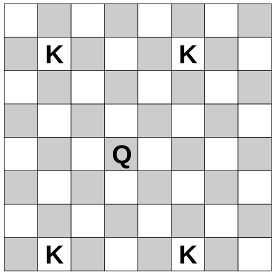
Greedy Solutions



Most attacking or most attacked.

- 5 pieces removed, 4 pieces remaining.

Actual Optimal Solution



- 4 pieces removed (minumum!), 5 pieces remaining.

By looking at special cases, we can find the rule that applies to a certain case, but that may not apply to every case.

Adjusting the Operators

Allowing different kinds of operators can help:

- In the previous example, the only operation allowed was to **remove** a piece. What if we allow **re-adding** a piece?
- If **re-adding** is allowed, solutions can be found where one **Q** can be added where there are four **K**s.
- Is there a greedy estimate function for this case?

Key Points

- DLS, IDS, BDS search order, expansions, and queueing
- DLS, IDS, BDS evaluation
- DLS, IDS, BDS: suitable domains
- Repeated states: why removing them is important
- Constraint Satisfaction Search: what kind of domains? why important?
- Best-first, greedy, and A^* search. How they differ.
- Why greedy search can fail?

Next Time (and Beyond)

- More informed search: Chapter 4
- A^*
- Heuristics
- Memory bounded search: Iterative deepening A^*
- Hill-climbing
- Simulated annealing